VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Optimalizace směrování dopravních vozidel

Optimize Routing of Transport Vehicles

2011 Radek Tomis

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. května 2011 .....

Rád bych poděkoval Ing. Janu Martinovičovi, Ph.D. za podporu a cenné znalosti, které pomohly tuto práci vytvořit.

Abstrakt

Tato práce se zabývá hledáním cest v silniční síti. Víme kam se chceme dostat, ale vybrat tu skutečně nejlepší cestu není jednoduché. Různé GPS navigace naleznou různé výsledky, protože záleží na použitých algoritmech a mapových podkladech. Za předpokladu, že mapové podklady jsou přibližně stejné, se zaměříme na možnost ovlivnění kvality vyhledávání algoritmicky. Díky velikosti silniční sítě klademe důraz na rychlost hledání. Klasický Dijkstrův algoritmus vylepšíme, optimalizujeme a představíme jeho novější varianty. Zaměříme se na obousměrné hierarchické vyhledávání a tvorbu rychlostní hierarchie z dostupných dat. V závěru porovnáme kvalitu vyhledávaných cest a rychlost výpočtů pro popsané optimalizace.

Klíčová slova: směrování, Dijkstrův algoritmus, nejrychlejší cesta, optimalizace, hierarchie

Abstract

This thesis deals with finding paths in the road network. We know where we want to, but picking the best way is not easy. Various GPS find different results, because it depends on used algorithms and maps. Assuming that maps are about the same, we focus on the possibility of influencing the quality of search algorithmic. Considering the size of the road network, we focus on the search speed. We improve classic Dijkstra's algorithm, optimize it and introduce the newer variants. We focus on bidirectional hierarchical search and creation of a highway hierarchy from available data. Finally, we compare the quality of found paths and computation speed for described optimizations.

Keywords: routing, Dijkstra's algorithm, fastest path, optimization, hierarchy

Seznam použitých zkratek a symbolů

GPS	–	Global Positioning System

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Hledání a plánování cesty trápí lidstvo od nepaměti. Kdysi, před érou počítačů, bylo nutné spolehnout se na odhad a zkušenosti. Dnes, narozdíl od našich předků, máme úplně jiné možnosti. GPS navigace jsou skoro ve všech moderních mobilních telefonech a hledání cesty je jednoduché. Přesto ale zařízení různých výrobců naleznou různé cesty. Podíváme se na příčiny, které vedou k rozdílnosti těchto výsledků.

Vycházíme z dříve vydané diplomové práce [9], která položila základ i pro tuto práci. Na rozdíl od ní se budeme zabývat optimalizačními technikami směrovacích algoritmů. Navážeme na již vytvořené části programové struktury a rozšíříme je o možnost vyhledávání tras nejvhodnějších z pohledu celkové doby jízdy, dále často označovaných jako nejrychlejší cesta. Implementace je v programovacím jazyce C# v prostředí .NET frameworku.

Obecně se na problém směrování podíváme v sekci 2. Popíšeme silniční síť, způsob, jakým ji transformujeme do programu a změny v původní struktuře, které musíme provést, abychom dále mohli směrovat i mezi Českou republikou a Slovenskem.

Sekce 3 obsahuje dva nejběžnější algoritmy pro vyhledávání cesty. Dijkstrův algoritmus [6] a A* [8] jsou základ pro mnoho moderních algoritmů, nicméně samotně bez dalších optimalizací se již prakticky nepoužívají.

V sekci 4 se budeme zabývat optimalizacemi, kterými lze vyhledávání urychlit. Probrané možnosti zahrnují změnu datové struktury pro reprezentaci prioritní fronty, heuristický přístup založený na odhadu a obousměrné hledání [12].

Využití hierarchie v sekci 5 popisuje nový přístup k rychlému směrování. Zavedeme pojem rychlostní hierarchie a provedeme kroky nutné k jejímu vytvoření. Dále představíme vlastní implementaci obousměrného hierarchického vyhledávání [7].

Sekce 6 je věnována dopravní dostupnosti. Ukážeme postup výpočtu z libovolného místa a určíme data, která k tomu potřebujeme.

Vliv nastavení různých parametrů na běh algoritmů a testy kvality vyhledávání jsou v sekci 7. Zahrnuty jsou věechny optimalizační techniky.

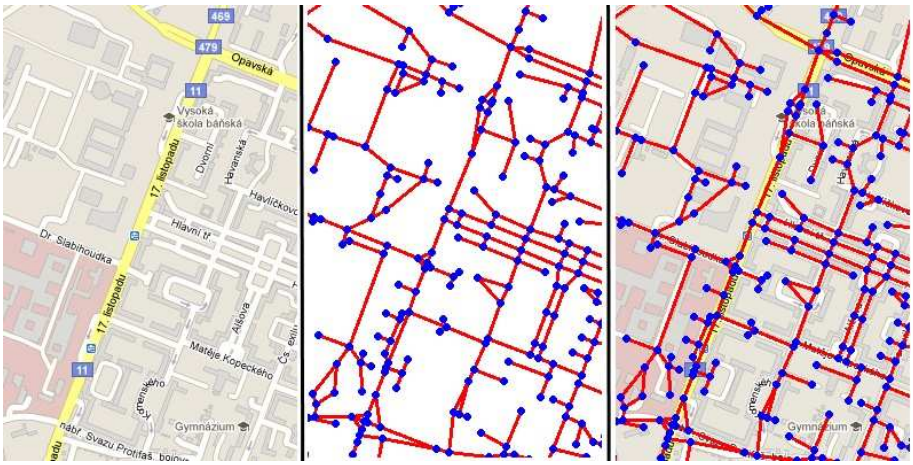
2 Směrování

Směrování je proces vybírání vhodné trasy, po které budeme vozidlo posílat. Probíhá v grafu [5], což je zjednodušený model nějaké reálné sítě, jako je například ta silniční. Graf obsahuje množinu vrcholů a hran, přičemž vrcholy znázorňují reálné objekty a hrany propojení mezi nimi. Směrování probíhá mezi dvěma vrcholy, zdrojovým a cílovým, a výsledkem je cesta mezi těmito vrcholy.

Základní problém směrování je nalezení nejkratší cesty v grafu s jedním zdrojovým a jedním cílovým vrcholem. Zabýváme se hledáním cesty mezi dvěma vrcholy, aby součet ohodnocení příslušných hran byl minimální. Tento problém řeší Dijkstrův algoritmus [6], popsaný dále v sekci 3.1.

2.1 Silniční síť

Silniční síť je forma orientovaného ohodnoceného grafu, který navíc obsahuje informace o geografickém umístění vrcholů a hran. Vrcholy představují křižovatky, zatáčky a jiné důležité objekty. Hrany jsou silnice nebo jen jejich části. Ohodnocení je nejčastěji tvořeno vzdálenostmi mezi vrcholy a je vždy nezáporné. Pro směřování v siliční síti se často používá algoritmus A* [8] popsaný v sekci 3.2, který efektivně využívá prostorové informace. Ukázka části silniční sítě je na obrázku 1.



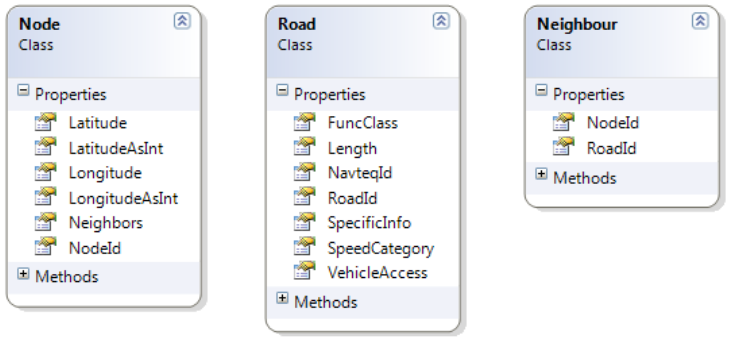
Obrázek 1: Silniční síť – vlevo mapa oblasti, uprostřed graf silniční sítě, vpravo spojení mapy a grafu

2.2 Struktura silniční sítě v programu

Dřívější práce [9] má vytvořenu strukturu a třídy pro směrování. Originální mapové podklady jsou uchovávány v databázi a obsahují nepřeberné množství dat. Stálé dotazování na databázi v průběhu algoritmu je velice pomalé, a proto byly vybrány jen nejnutnější hodnoty a vyexportovány do indexů v souborech.

Graf silniční sítě reprezentujeme v programu třídami *Node*, *Neighbor* a *Road*. Třídní diagram je na obrázku 2. *Node* představuje vrchol a obsahuje jednoznačné identifikační číslo, zeměpisnou šířku a výšku a pole sousedů (*Neighbor*). Soused je třída propojující daný vrchol s jiným vrcholem a nese informaci, kterou cestou (*Road*) se tam dostaneme. *Road* reprezentuje hranu a obsahuje jednoznačné číslo, délku úseku, rychlostní limit, kategorii a dvě značky. První značka určuje přípustná vozidla a rozeznáme s ní úseky pro nákladní vozidla a autobusy. Druhá značka obsahuje specifické informace, jako například, jestli se jedná o placenou silnici.

Dřívější struktura nepočítala s mezinárodním směrováním a neobsahovala informace o hraničních přejezdech. Proto byla třída *Road* rozšířena o příznak hraničního přejezdu. Dále vznikl soubor mapující vrcholy jedné strany hranice na vrcholy druhé strany. S tímto vylepšením můžeme vyhledávat mezi Českou republikou a Slovenskem, které jsme měli pro testování k dispozici. Navíc není problém v budoucnu přidat indexy dalších států a algoritmy na nich budou stále fungovat.



Obrázek 2: Třídní diagram programové struktury silniční sítě

2.3 Statické a dynamické sítě

Statické sítě vychází z jednoduchého předpokladu, že graf zůstává v čase neměnný. Silniční síť reprezentována mapovými podklady je tedy konstantní. Výhoda je v jednoduchosti a díky tomu se tento přístup používá nejčastěji. Naopak nevýhodou je nepřesnost výpočtů, pokud se některé části sítě mění.

Dynamické sítě obsahují prvky, které nejsou stálé. Do grafu podle potřeby vložíme nová data a ovlivníme tím směrování. Nejjednodušší příklad v silniční síti může být změna maximální rychlosti podle denní doby nebo zrušení průjezdnosti silnice při uzavírce. Dynamické sítě zajistí při dostatku aktuálních informací kvalitnější výsledek než sítě statické.

Směrovací algoritmy popsané dále jsou uzpůsobeny pro statické sítě.

3 Směrovací algoritmy

Směrovací algoritmy řeší problém nejkratší cesty v grafu s nezáporným ohodnocením hran, což silniční síť splňuje vždy. Směrování je založeno na cenách přechodu mezi dvojicemi uzlů. Cena je určena ohodnocením hrany a nižší hodnota značí kratší vzdálenost. Hledaná trasa musí mít minimální součet těchto cen.

Směrování je možné ovlivnit ohodnocením hran. Při hledání nejkratší cesty použijeme jako ohodnocení leteckou vzdálenost mezi uzly nebo skutečnou délku úseku. U hledání nejrychlejší cesty můžeme využít rychlostní limity, kategorii silnice nebo informace o dopravní situaci. Více dostupných dat znamená přesnější výsledek.

3.1 Dijkstrův algoritmus

Dijkstrův algoritmus [6] je grafový vyhledávací algoritmus řešící problém nejkratší cesty s jedním zdrojovým vrcholem. Algoritmus si uchovává množinu nalezených cest ze zdrojového uzlu do všech ostatních uzlů.

Popis algoritmu:

1. Nastavíme všechny vzdálenosti na nekonečno a vzdálenost zdrojového uzlu na 0.
2. Označíme každý uzel jako nenavštívený a zdrojový uzel jako aktuální.
3. Z aktuálního uzlu vypočítáme dočasné vzdálenosti do všech přilehlých uzlů, pokud je vzdálenost lepší než předešlá, přepíšeme ji.
4. Aktuální uzel označíme jako navštívený a jeho vzdálenost je konečná, tento uzel už nemusíme příště kontrolovat.
5. Pokud jsme navštívili všechny uzly, skončíme. V opačném případě vybereme z nenavštívených uzlů ten s nejmenší vzdáleností, označíme ho jako aktuální a pokračujeme od kroku 3.

Algoritmus projde každý uzel maximálně jednou a díky tomu je konečný. Navíc máme jistotu, že nalezené vzdálenosti jsou skutečně nejkratší. Pokud nás zajímá pouze cesta mezi dvěma uzly, můžeme vyhledávání ukončit, když narazíme na cílový uzel.

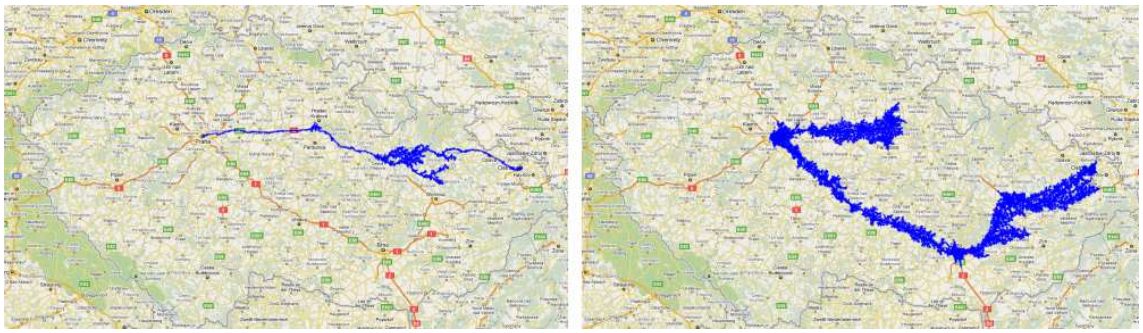
Místo vzdálenosti můžeme použít jakoukoliv jinou hodnotu. Jediné omezení je, že nižší ohodnocení vyjadřuje kratší cestu. Pro nejlepší cestu z pohledu celkové doby jízdy použijeme jako ohodnocení vzdálenost vydělenou rychlostí. Tak dostaneme čas potřebný k překonání daného úseku.

Dijkstrův algoritmus je z pohledu nalezené trasy optimální. Pro danou dvojici uzlů garantuje, že výsledná trasa je minimální a v grafu nenalezneme kratší. Daň za optimalitu je nutnost projít velké množství uzlů, a proto je hledání velice neefektivní a pomalé. Pro jakoukoliv implementaci množiny nenavštívených uzlů Q je časová složitost algoritmu $O(|E| \cdot sk_Q + |V| \cdot vm_Q)$, kde $|E|$ značí počet hran, $|V|$ počet uzlů, sk_Q a vm_Q je čas potřebný pro vykonání operace snížení klíče a vybrání minima z množiny Q. V sekci 4.1 se budeme více zabývat těmito operacemi.

3.2 A*

A* [8] je efektivnější verze Dijkstrova algoritmu, který rozšiřuje o heuristickou složku. Ohodnocení v každém uzlu se skládá z uražené vzdálenosti a odhadu ohodnocení zbývající cesty. Odhad je nejčastěji tvořen leteckou vzáleností k cíli. Pokud hledáme nejrychlejší cestu, musíme určit rychlost, se kterou vydělíme leteckou vzdálenost a dostaneme tak odhad doby cesty k cíli. Určená rychlost nesmí být příliš malá, protože by nadhodnotila odhad a algoritmus by vynechal některé možné cesy. Vybrání příliš vysokého odhadu rychlosti by naopak podhodnotilo ohodnocení a procházelo by se zbytečně mnoho nepravděpodobných cest.

Správné určení heuristické funkce má velký vliv na kvalitu nalezené cesty. Algoritmus nejdříve prochází směr, který nejpravděpodobněji vede k cíli a zároveň bere v potaz již uraženou vzdálenost. Nezaručuje tedy optimální výsledek, ale s přiměřeným nastavením odhadu rychlosti se k němu blíží. Množství prošlých uzlů je proti Dijkstrově algoritmu podstatně menší. Proto vyniká vysokou rychlostí výpočtu při přijatelné ztrátě přesnosti.



Obrázek 3: Prohledaná oblast A* pro zvolené rychlosti, vlevo $10ms^{-1}$, vpravo $20ms^{-1}$

Ukázka prošlé oblasti u výpočtu trasy Praha - Ostrava je na obrázku 3. Lze pozorovat, že nízko určený odhad rychlosti vytvoří téměř přímočarou cestu. Vyšší odhad projde větší oblast, ale blíží se optimálnímu výsledku. Vlivem heuristické složky na kvalitu vyhledávání se budeme dále zabývat v sekci 7.3.

4 Optimalizace směrovacích algoritmů

Optimalizační techniky směrovacích algoritmů v této sekci mají dopad na rychlost výpočtu. Kvalita nalezené trasy se ve většině případů také mění. Všechny optimalizace jsou zahrnuty do testů v sekci 7.

Moderní směrovací algoritmy využívají jako svůj základ některé z těchto optimalizací. Vývoj směřuje k předzpracovávání informací o grafu silniční sítě a následném využití těchto dat ke zrychlení samotného vyhledávání. Aktuálně nejrychlejsí algoritmus pojmenovaný Tranzitní uzly (Transit nodes) [3] předem zpracovává trasy mezi všemi důležitými uzly a ukládá si je do tabulky. K nalezení trasy potom stačí běžný Dijkstrův algoritmus, který provede hledání mezi důležitými uzly pomocí tabulky a zbytek cesty dopočítá. Nevýhoda tohoto přístupu je v době předzpracování, která je v rozmezí hodin až dnů. Pro účely této práce implementujeme svou verzi rychlého směrovacího algoritmu v sekci 5. Založíme ho na všech dále popsaných optimalizacích.

4.1 Prioritní fronta

Nejpomalejší část algoritmů, založených nebo odvozených od Dijkstrova algoritmu, je správa množiny nenavštívených uzlů. Operace, které nejčastěji provádíme jsou odebrání nejlépe ohodnoceného uzlu a přidání nově nalezených sousedů. Rychlost těchto operací znatelně ovlivňuje rychlost celého algoritmu. Typicky se pro tyto potřeby využívá prioritní fronta.

Prioritní fronta je nejběžnější typ abstraktní struktury, používané pro reprezentaci množiny neprozkoumaných uzlů. Jedná se o strukturu podobnou běžné frontě, ale každý prvek má navíc číselnou hodnotu, která určuje jeho prioritu. Prioritní fronta je zaměřena na rychlé vykonání příkazů vložení s určitou prioritou a odebrání prvku s nejvyšší prioritou.

Používané implementace pro reprezentaci prioritní fronty:

- Setříděné pole
- Přímo indexované pole
- Setříděný spojitý seznam
- Binární vyhledávací strom
- Binární halda

Konkrétní implementace prioritních front se liší v rychlosti jednotlivých operací. Výběr datové struktury závisí na požadovaných operacích a také množství jejich opakování. Dijkstrův algoritmus odebírá nejlépe ohodnocený uzel (minimum) pro každý přímo navštívený uzel. Testuje, jestli uzel už obsahuje u všech sousedů každého uzlu. Vkládá s prioritou nové uzly pro posouzené[1] sousedy. Navíc, pokud nalezne lepší ohodnocení pro už obsažený uzel, musí snížit prioritu.

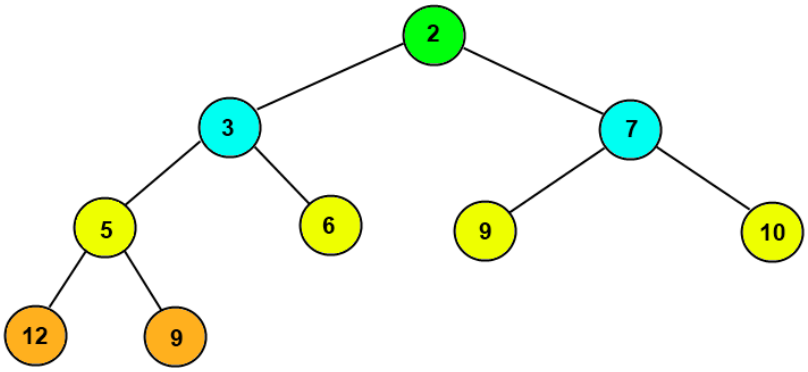
[1] posouzený soused je uzel, do kterého jsme z aktuálního uzlu nalezli lepší ohodnocení, než je současné

Existují i daleko pokročilejší struktury, než výše uvedené, jako například Fibonacciho halda, která by měla svými vlastnosmi předčít všechny ostatní implementace. Testy [11, 10] poukázaly ovšem na skutečnost, že teoretické rychlosti Fibonacciho haldy je dosaženo pouze u obrovských grafů. Pro menší grafy je mnohem rychlejší Binární halda, která pro naše potřeby předčí svými vlastnostmi ostatní možnosti.

4.1.1 Binární halda

Binární halda [2] je stromová datová struktura používaná pro reprezentaci množiny čísel nebo jiných porovnatelných hodnot. Kořen stromu obsahuje vždy minimální hodnotu a každý uzel má hodnotu menší nebo rovnu svým následovníkům. Každý prvek má maximálně 2 následovníky a přesně jednoho předka, pokud se nejedná o kořen. Z těchto pravidel si můžeme binární haldu představit jako obdobu binárního stromu, ve kterém jsou vždy oba následovníci větší, než aktuální uzel.

Výhoda binární haldy je v možnosti reprezentovat ji jako pole. U nejběžnější implementace platí, že v poli má prvek na indexu i své dva následovníky na pozicích $2i+1$ a $2i+2$, předchůdce na $\frac{i-1}{2}$ a minimální hodnotu na nulté pozici, kde se nachází kořen. Toto je možné vylepšit posunutím kořenu na první pozici. Potom prvek na indexu i má následovníky na pozicích $2i$ a $2i+1$ a předchůdce na $\frac{i}{2}$. Takto ušetříme dvě aritmetické operace. Násobení a dělení dvěma je dále možné provést jako bitový posun doleva nebo doprava o jedna. Obrázek 4 znázorňuje ukázku binární haldy a umístění jednotlivých hodnot podle indexu.



Hodnota	-	2	3	7	5	6	9	10	12	9
Index	0	1	2	3	4	5	6	7	8	9

Obrázek 4: Ukázka binární haldy a uspořádání hodnot pole podle indexu

Vkládání prvku probíhá následovně. Umístíme ho na poslední místo haldy, porovnáváme s jeho předchůdci a postupně prohazujeme směrem nahoru, než najdeme jeho

umístění. Odebrání funguje obdobně, poslední prvek přesuneme na pozici kořene a dokud nenajdeme správné umístění, tak porovnáváme s následovníky a prohazujeme směrem dolů. Tento postup zaručuje logaritmickou složitost těchto operací. Úprava hodnoty je pomalejší, protože ji musíme nejdříve najít mezi všemi hodnotami, a to znamená lineární složitost.

Signatura třídy reprezentující binární haldu je vidět níže. Generické typy TPriority a TValue usnadňují další práci. TPriority představuje typ priority, který je omezen na hodnotový typ kvůli rychlosti a musí umožňovat porovnávání vycházející z vlastností binární haldy. Jednu implementaci můžeme takto využít pro uchovávání priorit jako vzdálenost (celé číslo), čas (reálné číslo) nebo jakékoliv jiné hodnoty. Ke každé prioritě je asociována hodnota typu TValue. Dvojice jsou uloženy ve dvou samostatných dynamických polích.

```
class BinaryHeap<TPriority, TValue> where TPriority : struct, IComparable<TPriority>
```

Výpis 1: Signatura binární haldy

Zrychlení algoritmu díky binární haldě je otestováno v sekci 7.1.

4.2 Heuristické vyhledávání

První automobilové navigace se začaly prodávat v roce 1995. Výkon těchto zařízení byl slabý, a proto bylo nutné přizpůsobit vyhledávací algoritmus rychlosti hardwaru. Heuristické hledání je jedna z prvních používaných optimalizací, a to hlavně díky své jednoduchosti.

Heuristická metoda optimalizace zvyšuje výkon na úkor přesnosti. Není potřeba prohledávat nedůležité cesty, pokud nejsme blízko začátku nebo konci hledání. Důležité je správně určit, které silnice jsou důležité a které ne. Na správném odhadu závisí přesnost nalezené cesty. Tato optimalizace je závislá na mapových podkladech a pokud nemáme data umožňující určení důležitosti silnic, nelze optimalizaci použít.

Mapové podklady, které používáme pro směrování, obsahují data o kvalitách cest v rozmezí 1 až 5, přičemž kvalitativně nejlepší je úroveň 1, nejhorší 5, přehled i s počty úseků v ČR a na Slovensku naleznete v tabulce 1. Kvalita může sloužit jako dobrý ukazatel důležitosti dané cesty. Další možnost je použití rychlých cest. Přehled rozmezí rychlostních limitů a počet úseků je v tabulce 2.

Kategorie	Popis	Počet ČR	Počet Slovensko
1	Dálnice	7459	8380
2	Rychlostní silnice	35303	12888
3	Jiné významné silnice	83096	23726
4	Místní silnice	199966	47265
5	Ostatní	1245675	468232

Tabulka 1: Rozdělení silnic podle kategorií, počty úseků v ČR a na Slovensku

Rychlost	Počet ČR	Počet Slovensko
101 - 130	4616	2045
91 - 100	238	187
71 - 90	145392	37668
51 - 70	7782	6346
31 - 50	969307	333953
11 - 30	419017	173315
0 - 11	25147	6977

Tabulka 2: Rozdělení silnic podle rozmezí rychlostních limitů, počty úseků v ČR a na Slovensku

Tabulka 1 prozrazuje, že ve vyšších kategoriích (nižší číslo) je mnohem řídší pokrytí silnicemi. Musíme vytvořit heuristickou funkci, která nám umožní při hledání vynechat nedůležité uzly. Pro tento účel se hodí omezení kategorie silnic. Cesty vyšší kvality jsou potenciálně rychlejší. Předělat Dijkstrův algoritmus pro heuristické hledání je jednoduché, viz. výpis 2. Při průchodu směrovacího algoritmu přidáme kontrolu nově objevených uzlů na vzdálenost od zdrojového a cílového uzlu. Pokud je vzdálenost větší než určitá zadaná vzdálenost, ověříme kategorii silnice. Jestliže kategorie je nižší (vyšší číslo) než zadaná hodnota (limit), nepřidáme uzel do struktury a tím ho eliminujeme z vyhledávání. Vše výše popsané se vykoná jenom pokud nejsme na sjezdu z dálnice (fromMotorway).

```
if (!fromMotorway)
{
    if (!( DistanceFinder.FindDistance(startNodeGPS, node) <= length
        ||
     DistanceFinder.FindDistance(endNodeGPS, node) <= length ))
    {
        if (road.Category > limit) continue;
    }
}
```

Výpis 2: Heuristická metoda hledání

Chování hledání můžeme ovlivnit parametry length a limit. Length vymezuje oblast, ve které se prochází všechny uzly. Hodnota v rozmezí několika jednotek až desítek kilometrů je přijatelná a její velikost nehraje velkou roli. Limit určuje maximální kategorii, na které vyhledávání pokračuje mimo vymezenou oblast.

Test kvality trasy a času výpočtu pro vybraná nastavení naleznete v sekci 7.4.

4.3 Obousměrné vyhledávání

Obousměrné vyhledávání [12] je základní vylepšení vyhledávacích algoritmů. Funguje na principu dvou souběžných hledání. První postupuje směrem od počátečního uzlu ke koncovému (dopředné hledání), druhé od koncového k počátečnímu (zpětné hledání). Algoritmus skončí, když se tyto dvě vyhledávání potkají přibližně uprostřed.

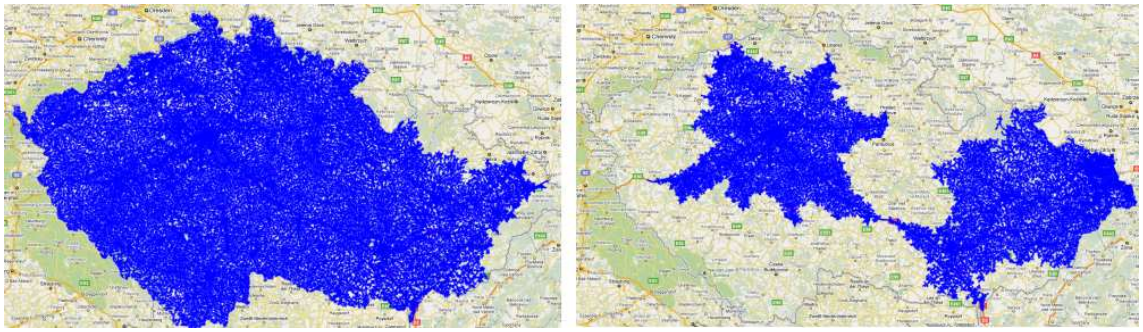
Implementace samotná není jednoduchá, jak by se mohlo na první pohled zdát. Musíme předělat logiku vyhledávání, aby bylo možné souběžně postupovat z obou stran. K tomu účelu si obě hledání musí uchovávat informace o navštívených i nenavštívených uzlech. Navíc u každého prošlého uzlu testujeme, jestli se nenachází v již navštívených uzlech protějšího hledání, pokud ano, nalezli jsme cestu. Toto testování se provádí pořád, a proto musíme použít strukturu, která umožňuje rychle ověřit členství, jako například hashovací tabulku.

Zrychlení je docíleno díky snížení množství uzlů, které je nutné projít. Vysoká rychlost (až dvojnásobná) a možnost kombinace s jinými optimalizačními technikami předurčují obousměrné vyhledávání k použití v dalších moderních vyhledávacích algoritmech. Některé pokročilé algoritmy, jako například hierarchické hledání v sekci 5.4, se bez obousměrného hledání dokonce neobejdou.

Testy obousměrného vyhledávání jsou v sekci 7.5.

4.3.1 Obousměrné vyhledávání s Dijkstrovým algoritmem

Podstata Dijkstrova algoritmu zajišťuje, že při průchodu expanduje rovnoměrně do všech směrů. V silniční síti je tato prohledaná oblast srovnatelná s kruhem. Rozložení na dvě části zajistí podstatné snížení prošlých uzlů (obsah dvou kruhů s poloviním poloměrem je proti celému kruhu přesně poloviční) a rychlost se díky tomu zvýší v ideálním případě až dvojnásobně. Malé zrychlení zajistí také využití výše popsané binární haldy, a to z toho důvodu, že dva menší stromy pracují nepatrně rychleji než jeden velký.



Obrázek 5: Prohledaná oblast - Dijkstrův algoritmus vlevo, obousměrný Dijkstrův algoritmus vpravo

Vizualizace prohledané oblasti u výpočtu trasy Praha - Ostrava je na obrázku 5. Dle předpokladů je oblast obousměrného vyhledávání přibližně poloviční proti běžnému Dijkstrovu algoritmu. Algoritmus neprochází uzly ve vzálenějších oblastech od počátku a konce hledání, jelikož nalezne trasu brzy a nestihne expandovat dále.

Obousměrné vyhledávání s Dijkstrovým algoritmem nezaručuje stejný výsledek, jaký bychom očekávali od běžného Dijkstrova algoritmu, protože je důležité pořadí procházených uzlů. Prohledaná oblast roste rovnoměrně z obou stran a při nalezení jediného uzlu protějšího hledání algoritmus končí. Díky postupnému střídání obou hledání nemusí být

nutně tento uzel na skutečně nejrychlejší cestě. Nalezené trasa se ale může lišit pouze o délku jednoho úseku silnice, a proto je rozdíl od správného výsledku minimální.

4.3.2 Obousměrné vyhledávání s A*

Využití obousměrného vyhledávání s A* algoritmem je velice problematické. Díky dvěma heuristickým složkám hrozí, že se hledání potkají ve špatném místě. Problém lze vyřešit globálním zmenšením heuristické složky v poměru k ohodnocení, čímž A* přiblížíme Dijkstrovu algoritmu. Tím algoritmus sice zpomalíme, ale bude hledat přesnější trasy.

Kvalita nalezené trasy není u tohoto algoritmu nikdy optimální. Heuristická složka má silný vliv na výsledek. V mnoha případech je nalezení trasy skoro náhodné. Algoritmus nicméně vyniká mezi všemi předešlými optimalizacemi po stránce výkonnostní. Pokud nám tedy nezáleží na kvalitě cesty a požadujeme vysokou rychlost výpočtu, použijeme právě obousměrné vyhledávání s A*.

5 Využití hierarchie

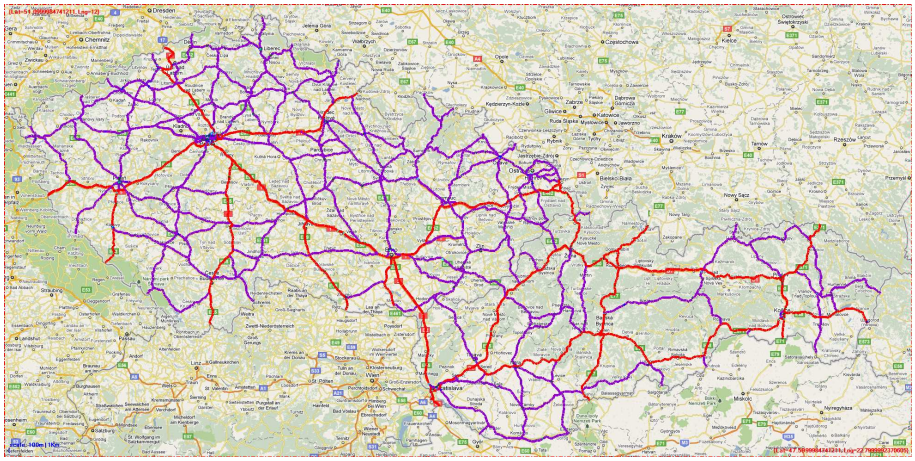
Silniční síť má specifické vlastnosti, které můžeme využít pro zrychlení vyhledávání. Jedná se především o kategorie silnic a rychlostní limity, které nám umožní najít nejrychlejší cestu. Část vyhledávání můžeme provádět pouze na silnicích vyšší kvality. V tabulce 1 je vidět, že počty úseků v kategoriích se rychle zmenšují s rostoucí kvalitou.

Využití hierarchie [7] předpokládá rozdělení kategorií na úrovně podle kvality. Úrovně můžeme určit jako jednotlivé kategorie nebo spojení několika kategorií.

Hledání cesty v hierarchickém grafu vyžaduje pro správné fungování souvislý (spojitý) graf propojený napříč jednotlivými úrovněmi. Tímto omezením zajistíme, že hledaná cesta musí existovat. Hledání postupuje z obou stran a vždy, když je to možné, přeskočí na cestu vyšší úrovně (kvalitnější). Tento přístup má nespornou výhodu, protože simuluje chování řidiče. Na začátku cesty se z místních komunikací snaží dostat na hlavní silnici, dále na silnici první třídy nebo i dálnici. Stejný postup se provede od konce hledání a na silnici nejvyšší úrovně uprostřed se vyhledávání potkají a cesta je nalezena. Podrobněji popíšeme naši verzi obousměrného hierarchického vyhledávání v sekci 5.4.

5.1 Rychlostní hierarchie

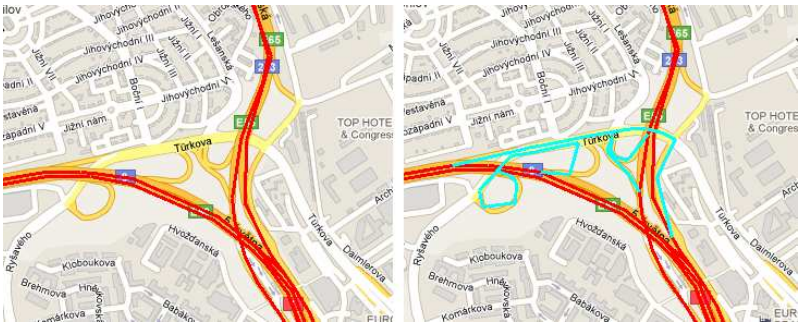
Pozorováním našich dat silniční sítě lze zjistit, že samostané kategorie nemusí tvořit souvislý graf. Tento problém je nejvíce patrný u dálničních nájezdů, které jsou často tvořeny silnicemi kategorie 4 (místní komunikace). Propojení dálnic je tvořeno stejným způsobem, a proto nejsou ve většině případů spojeny. Musíme si tedy z dostupných dat vytvořit vlastní rychlostní hierarchii. Experimentálně jsme spojili kategorii 1 a 2, čímž vznikne základ rychlostí úrovně. Ta je rovnoměrně rozložena po celé silniční síti a obsahuje pouze ty nejrychlejší cesty (obrázek 6).



Obrázek 6: Spojení silniční kategorie 1 a 2 na mapě České republiky a Slovenska

Vytvořený graf vypadá na první pohled souvisle, ale při větším přiblížení lze pozorovat nespojitosti a různé anomálie, ukázka na obrázku 7. Proto je nutné vymyslet způsob, kte-

rým by bylo možné tyto nespojitosti eliminovat. Nejjednodušší možnost spočívá ve využití klasického Dijkstrova algoritmu pro přepočítání nejrychlejších cest mezi sjezdovými uzly rychlostního grafu. V případě nalezení nové cesty je tato přidána do rychlostního grafu. Jelikož se jedná o časově velice náročnou operaci, musíme vytvořit určité optimalizace a úpravy.



Obrázek 7: Ukázka nespojitosti (vlevo) a cíl rychlostní úrovně (vpravo)

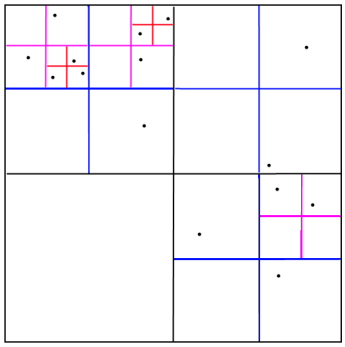
První optimalizace se týká omezení vzdálenosti, ve které se přepočet provádí. Zajímá nás pouze malá vzdálenost (maximálně několik kilometrů) mezi jednotlivými sjezdovými uzly. Provádět výpočet na větší vzdálenost může znamenat nalezení potenciálně rychlé cesty, ale není to výhodné z důvodu času potřebného pro běh algoritmu a také velikosti výsledného grafu, který je potom příliš veliký. V našich datech neexistuje rychlý způsob, jakým by bylo možné zjistit, zda jsou dané uzly od sebe do určité vzdálenosti. To znamená provést výpočet vzdálenosti v kvadratické složitosti, což je vzhledem k počtu uzlů velice neefektivní. Pro řešení tohoto problému je možné využít například kvadrantového stromu [4], který je popsán dále v sekci 5.2.

Dále potřebujeme, aby Dijkstrův algoritmus efektivně našel v oblasti původního grafu nejrychlejší cestu mezi daným sjezdovým uzlem a všemi ostatními. Algoritmus na vstupu dostane zdrojový uzel a seznam cílových uzlů. Pokud se mu v průběhu podaří nalézt všechny cesty, je ukončen a případné nové cesty jsou přidány do rychlostního grafu.

5.2 Kvadrantový strom

Kvadrantový strom [4] je datová struktura, která se používá k rozdělení plochy na části (kvadranty). Je navržen pro dvojrozměrný prostor, a proto i vkládaná data musí být nějakým způsobem pozicovatelná. Každý uzel v této stromové struktuře obsahuje přesně čtyři potomky a samotná data. Hlavní myšlenka spočívá v rekurzivním rozdělování kvadrantů při vkládání a následném přerozdělení již začleněných dat, ukázka na obrázku 8, stejné úrovně jsou barevně odlišeny. Předpoklad pro správné fungování je počáteční určení hraničních bodů a tím i vymezení oblasti.

Jednotlivé uzly mohou obsahovat jeden nebo více datových záznamů. Naše implementace (třídní diagram naleznete na obrázku 9, popis dále) toto určuje globálně pro všechny uzly. Jedná se o maximální počet záznamů, které může uzel pojmout. Po překročení dojde



Obrázek 8: Ukázka rozdělení prostoru kvadrantovým stromem

k vytvoření čtyř nových uzlů a záznamy jsou do nich přerozděleny podle své polohy. Takto se postupně data přesouvají do nižších vrstev stromu.

Největší přednost kvadrantového stromu je v rychlosti dotazování na nejbližší uzly v oblasti. Stromová struktura pro toto zajišťuje logaritmickou složitost, nicméně vzhledem k velikosti našeho grafu je nutné si ho přizpůsobit kvůli pamětové náročnosti.

Klasická implementace [13] vychází z použití tříd pro reprezentaci uzlů, přičemž každý uzel má čtyři reference na následnické uzly. Použití masivního množství tříd ale vyžaduje mnoho paměti i výkonu pro inicializaci. Stejně jako například jazyk C++ nabízí jazyk C# pro tento scénář využití struktury (struct). Použitím struktury dostaneme hodnotový typ, který obsahuje přímo své proměnné a množství zabrané paměti je proti třídě menší.

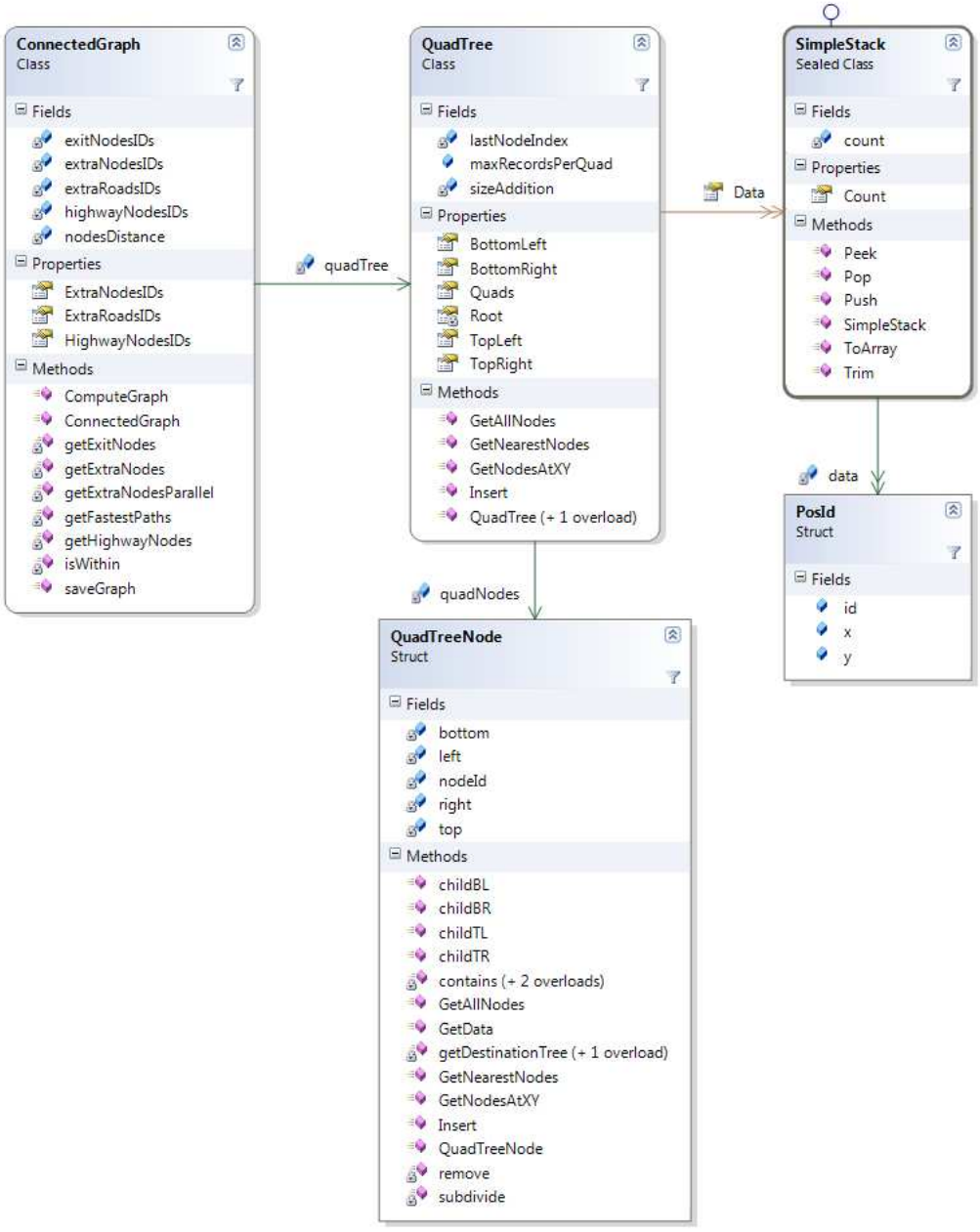
Problém nastává s logickým modelem kvadrantového stromu, konkrétně s referencemi. Struktura nesmí obsahovat proměnnou stejného typu, jako je ona sama, protože při deklaraci se vykoná implicitní konstruktor, čímž by vznikla lavinovitá inicializace a došla by pamět. Z tohoto důvodu musíme vytvořit svůj systém referencí uložený v polích. Potřebujeme proto samotnou třídu chovající se jako vstupní bod kvadrantového stromu, která bude tyto data uchovávat (třída *QuadTree* na třídním diagramu).

Uzly kvadrantového stromu (*QuadTreeNode*) obsahují jednoznačné identifikující číslo NodeId a údaje o své poloze. Nejdůležitější metoda je Insert , která vkládá záznamy typu *PosId*. Tento typ je struktura s proměnnými id, což je čislo uzlu v grafu a x, y určující pozici. *PosId* uzlu se ukládají do jednoduchého zásobníku (*SimpleStack*). Provázání uzlu s jeho daty je provedeno pomocí hlavní třídy *QuadTree*, která obsahuje slovník s klíčem rovným id uzlu a hodnotami typu *SimpleStack*. Přechod mezi jednotlivými uzly stromu je řešen čtyři celočíselnými poli TopLeft, TopRight, BottomLeft, BottomRight a polem se samotnými kvadrantovými uzly. Postup je následující:

1. Zjistíme id aktuálního uzlu.

2. V polích TopLeft, TopRight, BottomLeft a BottomRight jsou na pozicích s tímto id číselné záznamy, určující pořadí uzlů daného směru v poli kvadrantových uzlů QuadNodes.

3. Takto získáme samotné uzly, podle potřeby můžeme pokračovat s jednotlivými uzly znovu od prvního kroku.



Obrázek 9: Třídní diagram - spojitý graf, kvadrantový strom a pomocné třídy

Výsledný kvadrantový strom je pro naše účely velice rychlý a množství využité paměti zůstává i u velkého počtu uzlů v úměrných mezích.

5.3 Tvorba rychlostního grafu

Vytvoření rychlostního grafu zajišťuje třída *ConnectedGraph*. Metoda ComputeGraph přijímá na vstupu souřadnice, vymezující oblast výpočtu a velikost čtevrové oblasti, po které se postupuje.

Vytváření grafu v krocích:

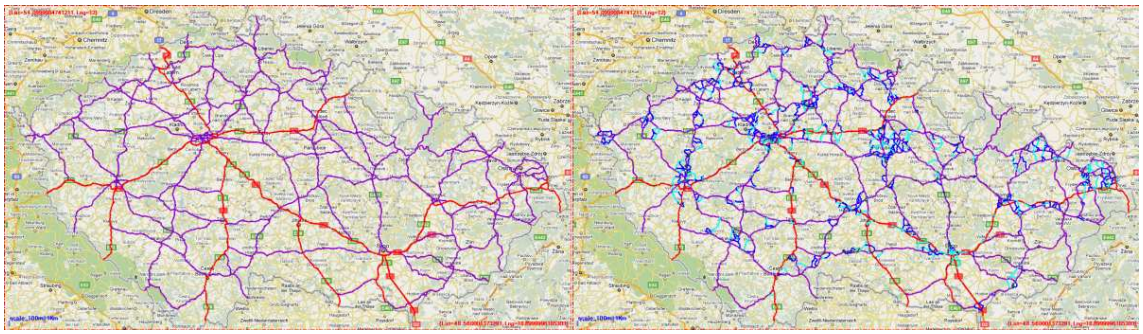
1. Pro každý uzel původního grafu ověříme, jestli se nachází na silnici kategorie 1 nebo 2. Pokud ano, uložíme si ho. Dostaneme základ rychlostní úrovně.

2. Zjistíme, ze kterých z těchto uzlů se můžeme dostat na silnice nižších tříd (kategorie 3 až 5) a uložíme si je jako sjezdové uzly do kvadrantového stromu.

3. Graf rozdělíme na čtvercové oblasti dle vstupní velikosti. V každé oblasti vybereme s pomocí kvadrantového stromu sjezdové uzly a provedeme mezi nimi výpočet nejrychlejších cest v celé silniční síti. Získané cesty si uložíme do seznamu přidaných uzlů.

4. Sjednocením základu rychlostní úrovně (bod 1) a přidaných uzlů (bod 3) dostaneme výslednou rychlostní úroveň.

Třetí krok je časově nejnáročnější část celého algoritmu. Výpočet v každé oblasti je ale zcela nezávislý, a proto je možné provést paralelně rozdělení na více vláken. Implementace k tomu účelu využívá součást .NET Frameworku nazývající se Task Parallel Library [1]. Knihovna obsahuje třídu *Task*, která představuje jednu dílčí úlohu, v našem případě výpočet v jedné čtvercové oblasti. Výhoda systému úloh spočívá v tom, že aplikace samotná rozhodne až za běhu, v jakém pořadí a na jakých vláknech se vykonání provede. Tím zajistíme maximální využití procesoru, bez ohledu na počet jeho jader.



Obrázek 10: Rychlostní úroveň pro ČR - porovnání výpočtu na 1 km (vlevo) a 10 km (vpravo)

Popsaný výpočet zajistí, že výsledná rychlostní úroveň je spojitá. Na obrázku 10 můžeme vidět vizualizace pro konkrétní výpočetní velikosti. Je zřejmé, že rychlostní úroveň obsahuje navíc rychlé cesty z nižších kategorií (zkratky). Díky přepočtu nejrychlejších cest máme jistotu, že tyto zkratky skutečně představují nejlepší trasy pro dané sjezdové

uzly. Výpočet na větší vstupní oblast zajistí vyšší výskyt takových zkratek. Počet je ale v porovnání s celou silniční sítí zanedbatelný.

Tabulka 3 znázorňuje čas výpočtu a počet vrcholů a hran pro vybrané vstupní velikosti. Nejkratší čas výpočtu je u velikosti 2 km. Vyšší čas výpočtu u menší vstupní velikosti je zapřičiněn rozdělením na příliš malé oblasti a následným provedením obrovského počtu jednoduchých hledání.

Velikost oblasti [km]	Počet uzlů	Počet hran	Čas výpočtu [s]
0,5	33254	35161	6,9
1	43556	45654	6,5
2	46824	49253	4,9
3	48605	51163	6,7
4	51506	54236	10,1
5	53161	56000	15,3
6	55654	58639	20,3
7	57474	62597	28,5
8	58311	63205	41,2
9	63243	67815	52,7
10	66712	71121	67,2

Tabulka 3: Počet uzlů a hran rychlostní úrovně a čas výpočtu pro vybrané vstupní velikosti

Gaf rychlostní úrovně pro další použití uložíme binárně do souboru ve formě všech unikátních identifikačních čísel uzlů a cest. Velikost souboru závisí na velikosti rychlostní úrovně a záleží na ni rychlost dále popsaného algoritmu v sekci 5.4. Musíme zvolit optimální poměr mezi velikostí vypočtené rychlostní úrovně a kvalitou hledání. V kategorii 7.7 je otestován vliv této velikosti na vyhledávání.

5.4 Obousměrné hierarchické vyhledávání

Obousměrné hierarchické vyhledávání vychází ze spojení obousměrného vyhledávání a rychlostní hierarchie. Algoritmus se skládá ze dvou souběžných hledání. Postupuje od zdrojového i cílového uzlu zároveň a hledá v okolí nejrychleji dostupnou silnici vyšší úrovně. Jakmile najde obě dvě, přeskočí na rychlostní graf a pokračuje v něm s vyhledáváním. Ve vyšší úrovni zůstává a není možnost přeskočení zpět. Tím zajistíme podstatné snížení prohledávaného grafu.

Nalezená cesta se vždy skládá z nájezdového úseku (od zdrojového uzlu po první uzel rychlostního grafu), následuje úsek rychlostní a nakonec sjezdový (ze sjezdového uzlu rychlostního grafu po cíl hledání). Toto pravidlo může být porušeno jen v jednom případě, a to že začátek i konec je velice blízko od sebe a nedojde k přeskoku na vyšší úroveň. Díky spojitosti, kterou jsme vytvořili v 5.3, je zaručeno, že cesta vždy existuje.

Proměnné použité v průběhu algoritmu jsou ve výpisu 3. OpenSet představuje množinu uzlů k prozkoumání. Jedná se o binární haldu popsanou v sekci 4.1.1, která používá aktuální cenu cesty jako prioritu a id uzlu jako hodnotu. ClosedSet je slovník navštívených

uzlů, klíčem je identifikační číslo uzlu a hodnotou pomocná třída *NodeTemp* obsahující informace o uražené vzdálenosti, času a odkud jsme se do uzlu dostali. Pomocný slovník HelpSet uchovává data o uzlu a nejlepší nalezenenou cenu, čímž urychluje ostatní operace. IsOnHighway určuje, jestli je hledání na rychlostním grafu. Tyto proměnné musí být vytvořena dvakrát, jednou pro každé z obou vyhledávání. Pro odlišení jsou dále označena velkým písmenem A (ze zdroje k cíli) a B (z cíle ke zdroji) na konci názvů proměnných.

```
BinaryHeap<double, int> openSet;
Dictionary<int, NodeTemp> closedSet;
Dictionary<int, double> helpSet;
bool isOnHighway;
```

Výpis 3: Proměnné používané u obousměrného hierarchického vyhledávání

Algoritmus začíná vložením počátečního a koncového uzlu do struktury. V průběhu hledání využíváme výstup výpočtu rychlostní úrovně ze sekce 5.3 v proměnné highwayGraph. Jedná se o hashovací tabulku umožňující rychlé ověření, zda se nacházíme na rychlostí úrovni. Následuje výpis 4 znázorňující základní část algoritmu.

```
int linkId = -1;

while (isActiveA || isActiveB)
{
  if (isActiveA)
  {
    linkId = dijkstra(endNode, openSetA, helpSetA, closedSetA, closedSetB, highwayGraph,
        isOnHighwayA);

    if (linkId != -1) break;
  }

  if (isActiveB)
  {
    linkId = dijkstra(startNode, openSetB, helpSetB, closedSetB, closedSetA, highwayGraph,
        isOnHighwayB);

    if (linkId != -1) break;
  }
}
```

Výpis 4: Popis obousměrného hierarchického vyhledávání

Tělo cyklu while se opakuje než skončí oba dílčí vyhledávání. Proměnné isActiveA a isActiveB určují, jestli můžou hledání pokračovat a jejich logická hodnota je odvozena od počtu záznamů v otevřené množině uzlů, tj. true pokud openSet obsahuje záznamy k prozkoumání. Obě části postupují souběžně a v každé iteraci expandují z každé strany jeden uzel pomocí metody s názvem dijkstra . Tato metoda funguje na základě Dijkstrova algoritmu a je rozšířena o práci s hierarchickým grafem. Pokud je proměnná isOnHighway nastavena na false, prochází celý graf a hledá uzly na rychlostním grafu. Pokud takový

najde, vymaže obsah openSetu a helpSetu, nastaví isOnHighway na true a začíná druhá fáze hledání, která se provádí jenom na rychlostním grafu. Do closedSetu se průběžně ukládají nalezené cesty. V každém volání metody dijkstra se ověřuje, jestli closedSet protějšího hledání neobsahuje aktuální uzel. Pokud ano, nalezli jsme cestu, vrátí se id tohoto uzlu, uloží se do proměnné linkId a hlavní cyklus je přerušen. Může se ale stát, že cesta neexistuje. To poznáme díky předem nastavené hodnotě -1 u linkId, jelikož po skončení algoritmu zůstane stejná.

Oba closedSety obsahují část nalezené trasy, a proto musíme výsledek převést do nějakého lepšího výstupního formátu. Použijeme k tomu pole třídy *ResultItem* (třída je popsaná v dřívější práci [9] - obsahuje souřadnice uzlu, uraženou vzdálenost a čas v konkrétním místě cesty). Z obou closedSetů dostaneme nalezené trasy zpětným průchodem, jelikož obsahuje prvky pouze s odkazem na jeden předchozí uzel, ze kterého jsme se do aktuálního dostali. Začneme z koncového uzlu obou hledání, což je společný prostřední bod a pokračujeme směrem k začátku hledání, tj. zdrojový nebo cílový uzel. Takto dostaneme dva pole typu *ResultItem*. Výsledek prvního hledání, ze zdrojového uzlu k cílovému, je v opačném pořadí než směr cesty a musíme ho převrátit. Druhé hledání začínalo z druhé strany a díky tomu jsou výsledky správně seřazené, ale hodnoty uražené vzdálenosti a času neodpovídají skutečnosti. Musíme je přepočítat a ke všem přičíst hodnoty z posledního *ResultItemu* prvního hledání. Výsledná cesta potom vznikne spojením výsledků prvního a druhého hledání za sebou.

Kvalita nalezené cesty je ve většině případů srovnatelná s Dijkstrovým algoritmem. Směrování na nejrychleji dostupný uzel rychlostní úrovně má ale za následek drobné odlišnosti na začátku a konci hledání. V nejhorším možném případě se může stát, že začátek cesty vede na opačnou stranu, než je směr skutečného cíle hledání. Proto byl algoritmus obohacen o nastavitelné parametry, se kterými dosáhneme lepších výsledků, zdrojový kód je na výpisu 5.

```
int length = tempNode.Length + road.Length;
double time = tempNode.Time + (road.Length / speed);
double distance = Math.Sqrt(
   Math.Pow(tempNode.Latitude − endNode.Latitude) +
   Math.Pow(tempNode.Longitude − endNode.Longitude));

double cost = length ∗ lengthFactor + time ∗ timeFactor + distance ∗ distanceFactor;
```

Výpis 5: Výpočet ceny přesunu do sousedního uzlu

Proměnná tempNode obsahuje aktuálně prohledávaný uzel. Vzdálenost (length) a čas (time) do sousedních uzlů vypočítáme jako součet aktuálních hodnot s hodnotami nového úseku cesty. Distance je letecká vzdálenost k cíli vypočítaná pomocí Pythagorovy věty. Cena přesunu do uzlu se potom rovná součtu těchto tří čísel upravených v poměru podle nastavených faktorů.

Algoritmus má dvě trojice těchto parametrů, jednu inicializační, která se uplatní na začátku a konci hledání, a druhou pro samotný rychlostní graf. Nastavení záleží na poža-

davcích. Pro nejrychlejší cestu využijeme inicializační parametry jako korekci při hledání nájezdových uzlů. Srovnání vlivu faktorů je v sekci 7.6.

6 Dopravní dostupnost

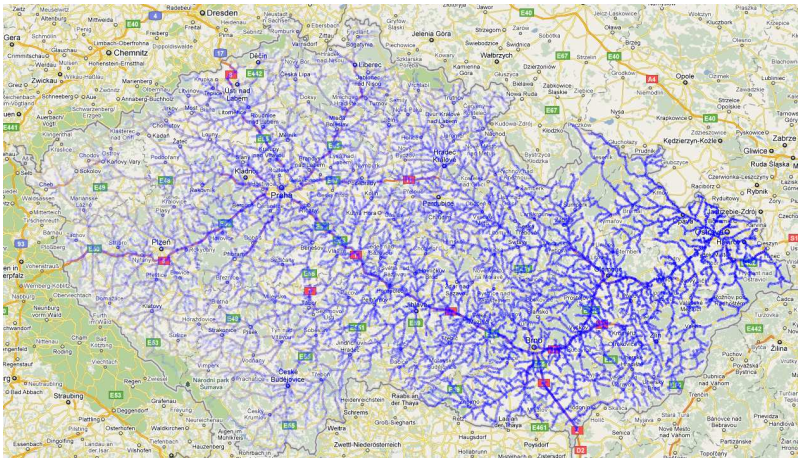
Dopravní dostupnost představuje v našem případě čas potřebný pro přesun z jednoho konkrétního bodu do všech ostatních významných míst v oblasti. Na grafu silniční sítě jsou důležité obce. Nynější data ovšem neobsahují potřebné informace (GPS souřadnice obcí), a proto bylo nutné najít náhradní zdroj. Nalezli jsme stránku[1] poskytující názvy obcí a jejich GPS souřadnice. Soubor pro Českou republiku obsahuje 6042 záznamů.

Výpočet dopravní dostupnosti provedeme s pomocí Dijkstrova algoritmu, který vyhledává z jednoho zdrojového uzlu do libovolně velkého seznamu cílových uzlů. Algoritmus vyhledává, dokud nenalezne všechny cílové uzly nebo neprojde celý graf. Jedním průchodem dostaneme trasy mezi zdrojovým uzlem a všemi ostatními. Převod GPS souřadnic obcí na uzly grafu provedeme pomocí kvadrantového stromu popsaného v sekci 5.2.

Průběh výpočtu pro daný zdrojový uzel

1. Naplníme kvadrantový strom
2. Načteme GPS souřadnice obcí
3. Kvadrantovým stromem přiřadíme uzly k souřadnicím
4. Spustíme algoritmus mezi zdrojovým uzlem a uzly představujícími obce
5. Algoritmus ke každému vstupnímu uzlu přiřadí vzdálenost a čas

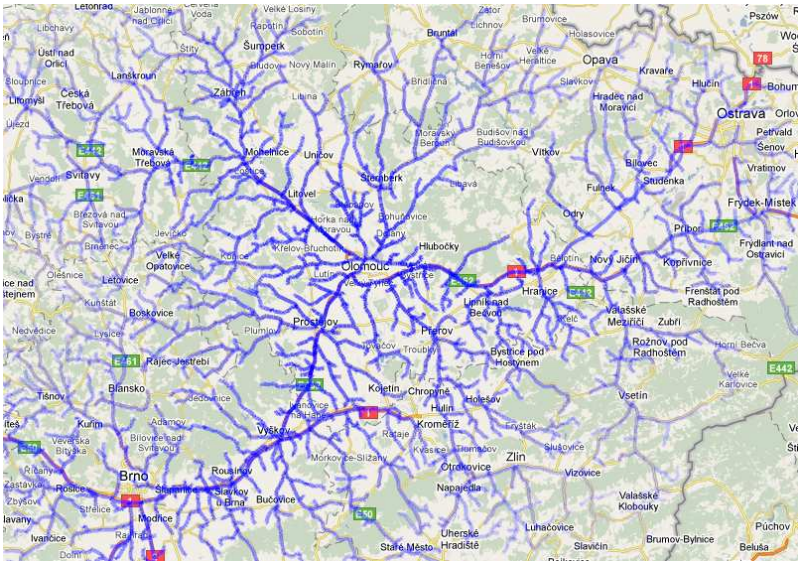
Dopravní dostupnost obcí je pouze jedna konkrétní možnost výpočtu. Algoritmus může pracovat s libovolným seznamem vstupních souřadnic a vypočítat k nim jejich dostupnost. Pro zpracování všech výsledků stačí pouze jeden průchod algoritmu. Počet prošlých uzlů je v nejhorším případě stejný, jako počet všech uzlů silniční sítě.



Obrázek 11: Vizualizace dopravní dostupnosti se zdrojovým uzlem v Ostravě

[1] geonames.org - geografická databáze

Výsledek je vzdálenost a doba cesty do všech obcí. Vizualizace výsledku se zdrojovým uzlem umístěným v Ostravě je na obrázku 11. Sytost modré barvy značí časovou dostupnost v daném místě. Kolem rychlostních silnic je dle očekávání dostupnost nejvyšší. Vizualizace dopravní dostupnosti pro Olomouc a její blízké okolí je na obrázku 12.



Obrázek 12: Vizualizace dopravní dostupnosti se zdrojovým uzlem v Olomouci

7 Testy

Následující sekce je věnována testům popsaných algoritmů. Testovací trasy u měření kvality algoritmů jsou vybrány pro následující vlastnosti.

- Aš Ubĺa - jedna z nejdelších tras mezi ČR a Slovenskem
- Pardubice Opava - méně rychlostních silnic v oblasti
- Břeclav Šluknov - dlouhá trasa s možností využití dálnic
- Jihlava Třeboň - krátká trasa mimo hlavní rychlostní silnice
- Vysočany Svitavy - průměrná trasa mezi dvěmi malými obcemi

Rozdílnost výsledků se často projeví pouze na větší vzdálenosti, nebo když jsou kvalitnější silnice dále od začátku hledání. Na konci sekce je souhrn všech měřených algoritmů.

Testy jsou provedeny na počítači s operačním systémem Windows 7 64-bitů. Použitý procesoror je Intel Core i5-750 s taktem 2,66 GHz, operační paměť 4 GB. Pro větší přesnost byla vypnuta funkce Turbo Boost a každý test běžel pouze na jednom jádru.

7.1 Vliv prioritní fronty

Test porovnává, jaký vliv má prioritní fronta na čas výpočtu klasického Dijkstrova algoritmu. Základní implementace (DA) používá setřízený slovník (*SortedDictionary*), který vnitřně funguje jako binární vyhledávácí strom, a vylepšená verze (DA BH) binární haldu popsanou v sekci 4.1.1.

	Čas výpočtu [ms]		
Trasa	DA	DA BH	Zrychlení [%]
Aš - Ubĺa	15509	12251	21,1
Opava - Pardubice	13034	8549	34,4
Břeclav - Šluknov	12963	9798	24,4
Jihlava - Třeboň	6713	3662	45,4
Vysočany - Svitavy	8265	4541	45,1

Tabulka 4: Vliv implementace prioritní fronty na čas výpočtu

Vliv implementace prioritní fronty je z tabulky patrný. Binární halda zvyšuje rychlost algoritmu u měřených tras o 21% až 45%. Bez jakékoliv změny v interní logice vyhledávání zvýšíme rychlost pouze změnou datové struktury. Nejvyššího zrychlení je dosaženo u kratších tras s menším počtem prozkoumaných uzlů. Všechny další optimalizace se snaží o snížení počtu nutně prozkoumaných uzlů a zrychlení může být tedy ještě vyšší. Proto je dále binární halda použita ve všech testovaných algoritmech.

7.2 Srovnání Dijkstrova algoritmu s webovými plánovači tras

Srovnání s webovými plánovači tras je důležité pro posouzení celkových výsledků. Jako zástupce kvality našich výsledků použijeme Dijkstrův algoritmus. Porovnáváme s webovými plánovači Seznamu[1] a Googlu[2]. Nastavení je pro všechny stejné. Vyhledáváme nejrychlejší cestu a jsou povoleny dálnice i placené úseky.

	Doba cesty [s]		
Trasa	DA	Google	Seznam
Aš - Ubĺa	43253	40740	40320
Opava - Pardubice	12754	11300	9600
Břeclav - Šluknov	16264	16080	14280
Jihlava - Třeboň	6517	4800	5880
Vysočany - Svitavy	10546	7920	8700

Tabulka 5: Srovnání Dijkstrova algoritmu s webovými plánovači tras

Doba cesty našeho Dijkstrova algoritmu je podstatně vyšší než u webových služeb. To ale neznamená, že by byly výsledky špatné. Důležitý faktor je, že používáme pouze mapy České republiky a Slovenska. Oba webové plánovače nalezly trasu Aš Ubĺá přez Německo a Polsko a začátek trasy Břeclav Šluknov po krátkém úseku Německé dálnice. Bez mapových podkladů nemůže náš algoritmus tyto cesty nalézt. Zbylé tři trasy byly u všech tří možností vyhledávání totožné. Znamená to, že rozdílnosti ve výsledcích jsou zapřičiněny pouze způsobem vypočítání celkové doby cesty. Náš algoritmus používá pesimistický přístup, u kterého znevýhodňujeme silnice nižší kvality. Předpokládáme jízdu nižší rychlostí než je rychlostní limit. Oba plánovače používají optimističtější přístup a jejich doba cesty je tedy nižší.

7.3 Vliv heuristické složky A*

Heuristická složka A* je rychlost v metrech za sekundu, kterou použijeme pro výpočet odhadovaného času cesty k cíli. Nastavení ovlivňuje poměr heuristické složky k uražené době a mění tak chování algoritmu. V tabulce 6 porovnáváme A* pro hodnoty rychlosti 10, 15 a 20 ms^{-1} s Dijkstrovým algoritmem, který můžeme také brát jako A* s heuristickou složkou 0.

Výsledky jdou porovnávat po řádcích. Sloupce jsou seřazeny záměrně, aby byly rozdíly patrné. Čas výpočtu se s klesající rychlostí (vyšší heuristické složce) razantně snižuje. Zároveň ale doba cesty stoupá. A* 10 je ukázkový příklad velice rychlého nastavení, které je v praxi nepoužitelné, protože nachází nekvalitní trasy. Hodnoty nižší než 10 hledají už jen přímočarou cestu. Naopak hodnoty vyšší než 20 se pomalu přibližují k Dijkstrovu algoritmu.

[1] mapy.cz
[2] maps.google.cz

Trasa	Čas výpočtu [ms]				Doba cesty [s]			
	DA	A* 20	A* 15	A* 10	DA	A* 20	A* 15	A* 10
Aš - Ubĺa	12251	11574	2844	1349	43253	43923	45676	48523
Pardubice - Opava	8549	1739	896	217	12754	12760	12761	14465
Břeclav - Šluknov	9798	3257	1467	348	16264	16367	16602	19105
Jihlava - Třeboň	3662	498	281	22	6517	6520	6587	6720
Vysočany - Svitavy	4541	1245	812	150	10546	10584	10549	10647

Tabulka 6: Vliv heuristické složky A*

7.4 Vliv omezení heuristického vyhledávání

Heuristické vyhledávání v sekci 4.2 umožňuje zvýšení výkonu vynecháním nepodstatných silnic. Pro porovnání je využit Dijkstrův algoritmus. V názvu je číslo heuristického limitu, určující, do které úrovně se prochází uzly mimo vymezenou oblast. Limit 1 není možné použít, protože dálniční síť je příiš řídká a ve většině případů není cesta nalezena. Limit 5 je ekvivalentní s neupraveným Dijkstrovým algoritmem.

Trasa	Čas výpočtu [ms]				Doba cesty [s]			
	DA	DA H4	DA H3	DA H2	DA	DA H4	DA H3	DA H2
Aš - Ubĺa	12251	2809	1157	305	43253	44309	44540	46876
Pardubice - Opava	8549	2303	1131	610	12754	12911	13080	13596
Břeclav - Šluknov	9798	2410	922	418	16264	16350	16350	17093
Jihlava - Třeboň	3662	1166	573	338	6517	6520	6520	6555
Vysočany - Svitavy	4541	1290	590	273	10546	10551	10656	10656

Tabulka 7: Vliv omezení heuristického vyhledávání

Nízký limit (vysoká heuristika) značně urychluje vyhledávání, ale také zvyšuje dobu cesty. Naopak vyšší limit prohledává větší oblast, díky tomu je přesnější, ale i pomalejší. Využití heuristického hledání je přínos při nedostatku výpočetního výkonu. Pokud máme data silniční sítě, ve kterých jsou jednotlivé úrovně alespoň částečně spojité, je implementace velice jednoduchá. Toto je ovšem podmínka pro správnou funkci. Malé nespojitosti zaviní nekvalitní výsledek.

7.5 Vliv obousměrného vyhledávání

Test porovnává obousměrný Dijkstrův algoritmus (ODA) ze sekce 4.3.1 a obousměrný A* (OA*) 4.3.2 s nastavenou rychlostí 10 a 20 ms^{-1}. V tabulce 8 můžeme vidět, jaký vliv má hledání z obou konců trasy zároveň.

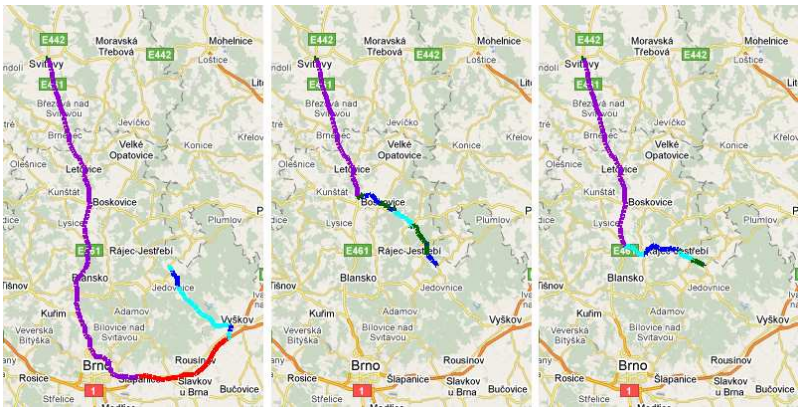
Obousměrný Dijkstrův algoritmus ve všech trasách zrychluje hledání a výsledek je jen nepatrně delší než u běžného Dijkstrova algoritmu. Obousměrný A* je ale ještě mnohem rychlejší. S nastaveným parametrem 20 je doba cesty pořád přijatelná, parametr 10 však už najde velice špatné trasy.

Trasa	Čas výpočtu [ms]				Doba cesty [s]			
	DA	ODA	OA* 20	OA* 10	DA	ODA	OA* 20	OA* 10
Aš - Ubĺa	12251	10073	3202	367	43253	43255	43661	49774
Pardubice - Opava	8549	3371	988	82	12754	12807	13223	14448
Břeclav - Šluknov	9798	3689	646	100	16264	16437	16446	22265
Jihlava - Třeboň	3662	916	223	17	6517	6553	6565	6603
Vysočany - Svitavy	4541	1642	577	98	10546	10546	10605	13791

Tabulka 8: Vliv obousměrného vyhledávání

7.6 Vliv inicializačních faktorů obousměrného hierarchického vyhledávání

Obrázek 13 porovnává tři možnosti nastavení inicializačních faktorů. Vliv je patrný na první pohled. Upřednosnění pouze rychlosti má za následek nalezení delší cesty. Hledání přímočaré cesty je lepší, ale algoritmus zůstává po delší dobu na silnicích nižší kategorie, a tím bude výsledná cesta pomalejší než by měla být. Kombinací rychlosti a přímočarosti dosáhneme ideálního výsledku.



Obrázek 13: Vliv inicializačního faktoru na vyhledání nájezdové silnice, vlevo nejrychlejší, uprostřed nejpřimočařejší, vpravo kombinace.

Kombinaci provedeme podobně jako v A* algoritmu složením ohodnocení z uplynulého času a odhaduté doby cesty k cíli. Pokud timeFactor nastavíme na 1, musíme distanceFactor nastavit na 1 děleno určená rychlost, aby byly obě složky ve správném poměru. Takto si algoritmus najde uzel rychlostní hierarchie, který se jeví jako ideální z pohledu času i polohy.

Všechny další testy obousměrného hierarchického hledání využívají kombinaci. Rychlost nastavíme po předchozích testech na 20 ms^{-1}.

7.7 Vliv velikosti rychlostního grafu

Rychlostní graf vypočtený v 5.3 svou velikostí ovlivňuje hledání. V tabulce 9 je porovnán Dijkstrův algoritmus s obousměrným hierarchickým vyhledáváním (OHV), které používá rychlostní graf vypočítaný na velikost 1, 5 a 10 km.

	Čas výpočtu [ms]				Doba cesty [s]			
Trasa	DA	OHV10	OHV5	OHV1	DA	OHV10	OHV5	OHV1
Aš - Ubĺa	12251	362	307	272	43253	43730	43738	43771
Pardubice - Opava	8549	138	104	81	12754	13479	13511	13673
Břeclav - Šluknov	9798	168	125	244	16264	16472	16500	16514
Jihlava - Třeboň	3662	31	28	26	6517	6543	6723	6724
Vysočany - Svitavy	4541	57	44	40	10546	10746	10746	10916

Tabulka 9: Vliv velikosti rychlostního grafu na kvalitu hledání

Tabulka prozrazuje, že s rostoucí velikostí rychlostního grafu stoupá čas výpočtu a zároveň se snižuje doba cesty. Správná volba velikosti grafu neexistuje. Pokud vyžadujeme rychlost, je lepší malý. Naopak jestliže je kvalita výsledku prioritní, použijeme větší. Nalezené trasy jsou ale ve všech případech velice přesné a rozdíly pro různé velikosti jsou malé.

7.8 Souhrnný přehled

Souhrnný přehled testů je na grafu 14. Výsledky daného algoritmu jsou zprůměrovány pro všechny trasy a relativně porovnány s běžným Dijkstrovým algoritmem (DA). Ukazatel kvality vyhledané trasy je procentuálně vyjádřené prodloužení doby cesty. Zrychlení algoritmů je po optimalizacích příliš vysoké, a proto je uvedeno v tisících procent. Navíc takto můžeme porovnávat hodnoty na stejném rozsahu grafu s tím, že rychlost je tisíckrát méně podstatná v poměru s kvalitou.

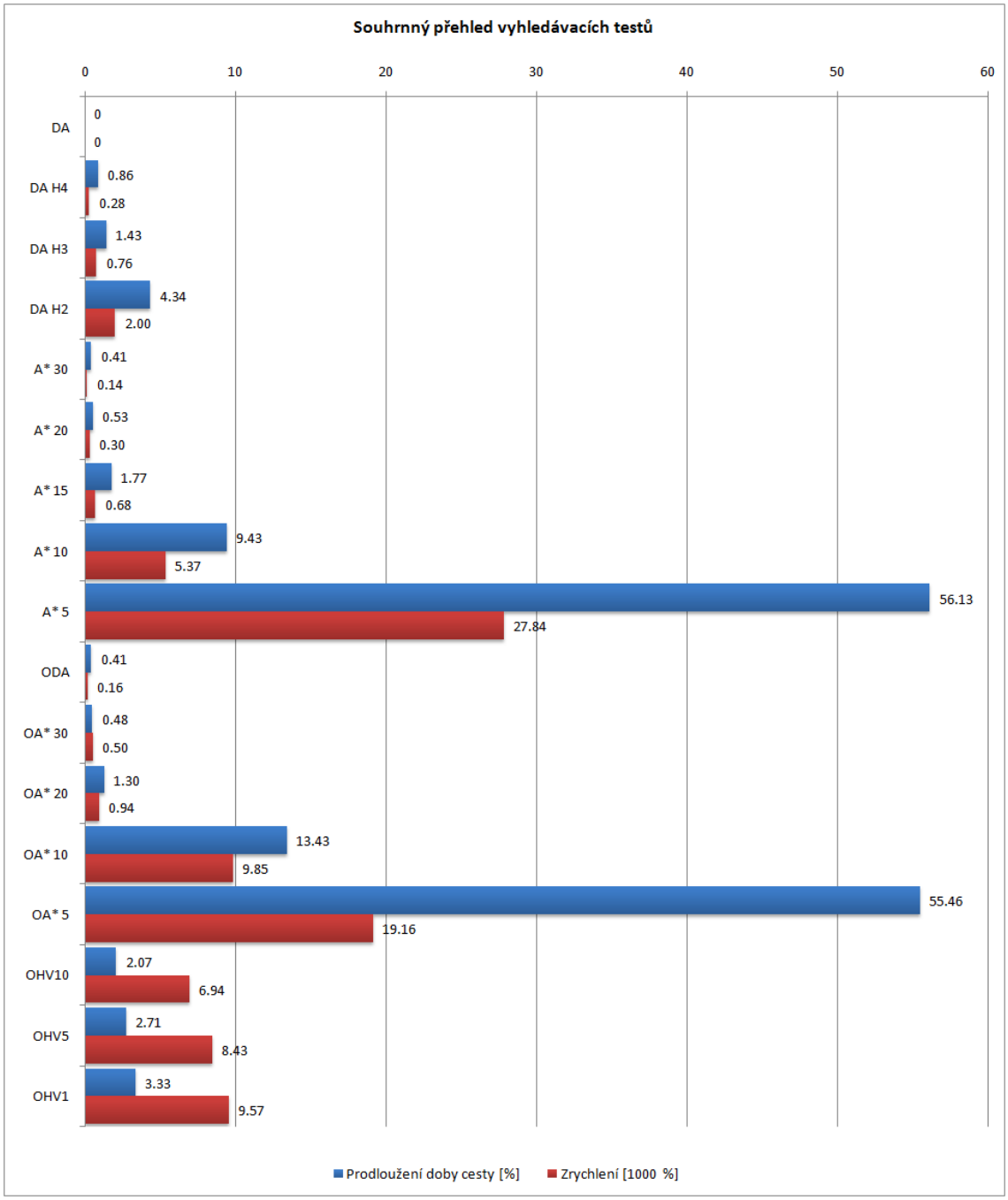
Heuristické vyhledávání (DA H) je nevýrazná optimalizace, která lehce zvýší rychlost výpočtu, ale i prodloužení trasy. Kvalita přímo závisí na mapových podkladech a v našem případě můžeme použít pouze čtyři možnosti nastavení algoritmu.

A* může změnou heuristické složky podstatně ovlivňovat hledání. V našem měřítku je prodloužení přibližně dvojnásobné proti zrychlení. Nejedná se o příliš dobrý výsledek, ale využití algoritmu je v jeho dobré nastavitelnosti.

Obousměrný Dijkstrův algoritmus (ODA) je ze všech optimalizací nejpřesnější a v kombinaci s nepatrným zrychlením výpočtu se z něj stává vhodná alternativa k běžnému Dijkstrovu algoritmu.

Obousměrný A* (OA*) vyhledává obodbně jako běžný A* velice nepřesné trasy. Rychlostně jsou na tom oba algoritmy přibližně stejně.

Obousměrné hierarchické vyhledávání (OHV) má nejvyšší poměr zrychlení k prodloužení. S vysokou rychlostí hledá přesné cesty. Díky tomu se jedná o nejlepší z testovaných algoritmů. Velká výhoda je navíc, že velikostí rychlostního grafu můžeme měnit kvalitu trasy.



Obrázek 14: Souhrn testů - graf prodloužení doby cesty a zrychlení proti Dijkstrovu algoritmu

8 Závěr

Každá optimalizace si vybírá daň v podobě prodloužení doby cesty. Pro optimální výsledek bychom museli použít Dijkstrův algoritmus, který vyžaduje vysoký výpočetní výkon. Nicméně běžné navigace nemají tento výkon k dispozici a musí na úkor kvality použít některý z typů optimalizace.

Optimalizace změnou prioritní fronty na binární haldu se ukázala jako důležité vylepšení, které urychluje všechny další algoritmy. Podstatné zrychlení je pozorovatelné při procházení velkého množství uzlů, a je tedy nejvyšší u Dijkstrova algoritmu. Ostatní optimalizace těží méně z využití binární haldy, protože se samy o sobě snaží projít co nejmenší počet uzlů.

Obousměrné hierarchické vyhledávání v této práci vychází z dříve popsaných teoretických základů. Jedná se o spojení myšlenek heuristického vyhledávání, A* a obousměrného vyhledávání. Referenční implementace využívá pouze A* a počítá s několika úrovněmi hierarchie. Naše implementace používá k vyhledávání nájezdových uzlů A* a dále na rychlostní úrovni vyhledáváme pomocí Dijkstrova algoritmu. Úprava algoritmu nám umožňuje měnit parametry vyhledávání (nejkratší trasa, nejrychlejší, nejpřímočařejší nebo kombinace). Kvůli problémům s nespojitostí kategorií silniční sítě jsme použili pouze dvě úrovně hierarchie, které jsme si navíc museli vytvořit. Naše vlastní implementace pro vytvoření spojité rychlostní úrovně je výhodná hned ze dvou důvodů: máme jistotu, že tvoří souvislý graf a obsahuje zkratky. Díky tomu dosahujeme přesnějších výsledků než referenční implementace. Navíc můžeme podle potřeb použít různé velikosti rychlostních úrovní a měnit tak kvalitu výsledné trasy.

Testy ukázaly možnosti optimalizovaných směrovacích algoritmů. Vliv nastavení na vyhledávání nám pomohl porozumět chování algoritmů za daných podmínek. Obousměrné hierarchické vyhledávání dosahuje ideálních výsledků z pohledu času výpočtu i kvality cesty. Vysoká rychlost a také přesnost nalezené trasy nemá mezi ostatními optimalizacemi konkurenci. Na druhou stranu je ale nutné před samotným hledáním vytvořit rychlostní úroveň. Tato operace je relativně rychlá a stačí ji provést pouze jednou.

Můžeme konstatovat, že investice času pro předzpracování rychlostní úrovně se vyplatí a je to jen malá cena za zrychlení a zpřesnění všech dalších vyhledávání.

9 Reference

[1] Task parallel library. http://msdn.microsoft.com/en-gb/library/dd460717.aspx.

[2] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 1986.

[3] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 2007.

[4] J. L. Bentley and R. A. Finkel. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 1974.

[5] Béla Bollobás. *Modern graph theory*. Springer, 1998.

[6] Edsger W. Dijkstra. *A note on two problems in connexion with graphs*. Numerische Mathematik, 1959.

[7] A. Efentakis, D. Pfoser, A. Voisard, and C. Wenk. Exploiting road network properties in efficient shortest-path computation. Technical report, ICSI, 2009.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael. *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. SIGART, 1972.

[9] Petr Havíček. Algoritmy pro směrování dopravních vozidel. Master's thesis, VŠB - Technická univerzita Ostrava, 2010.

[10] Donald E. Knuth. *The Stanford GraphBase*. ACM Press, 1994.

[11] Amit Patel. Amit's a* pages. http://theory.stanford.edu/~amitp/GameProgramming, 2010.

[12] Ira Pohl. Bi-directional search. *Machine Intelligence*, 1971.

[13] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 1984.